Generating Bus Traffic Patterns

Jacob Sander Andersen, CTO, SyoSil ApS, jacob@syosil.com
Lars Viklund, Expert Engineer, Axis Communications AB
lars.viklund@axis.com
Kenneth Branth, Senior Consultant, SyoSil ApS, kenneth@syosil.com

accellera
SYSTEMS INITIATIVE

2018
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE

The Challenge in Short

- Verification engineer required to apply a specific traffic pattern on a hardware bus like AMBA AXI

- Why?
  - Functional requirements (e.g. performance requirements)
  - Verification closure (e.g. metrics like structural coverage)

Motivation

- Initial scenario…
  - Designer:
    - Answers questions ☺
  - Verification engineer:
    - Asks questions
    - Takes notes in informal language
    - Does the implementation
- 2nd try…
  - Designer:
    - Answers questions ☺
  - Verification engineer:
    - Asks questions
    - Defines transaction diagram
    - Does the implementation

Core Problem, Consequence & Solution

- Communication between designer and verification engineer
  - Information loss
  - Leading to incorrect implementation

- Consequence
  - Many iterations
  - Loss of productivity (Wasting designer and verification engineers time)

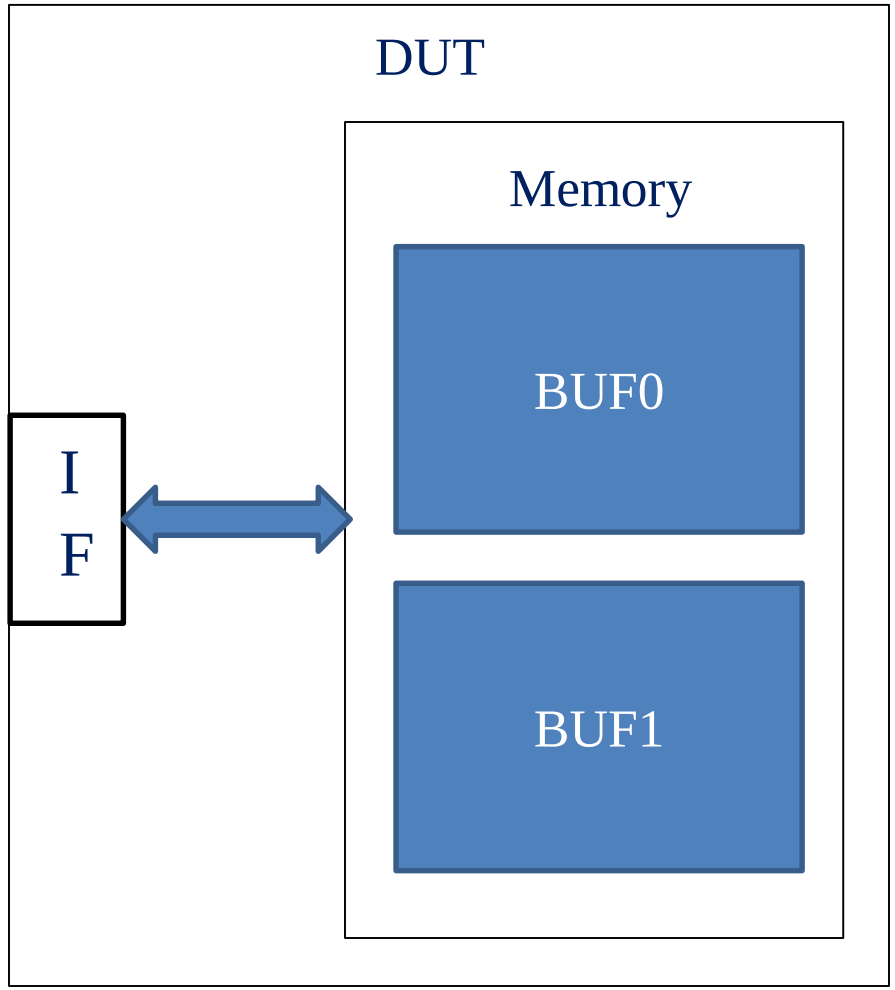
- Solution
  - Let the designer express the traffic pattern directly and accurately

But wait!!!

- Is this not just PSS?
- Both yes and no..
  - Some traffic patterns could be captured by PSS
  - This is targeting fine grained control of transactions on a single (or multiple) interfaces
    - Very specific scenarios for coverage closure
    - Very specific scenarios for performance verification

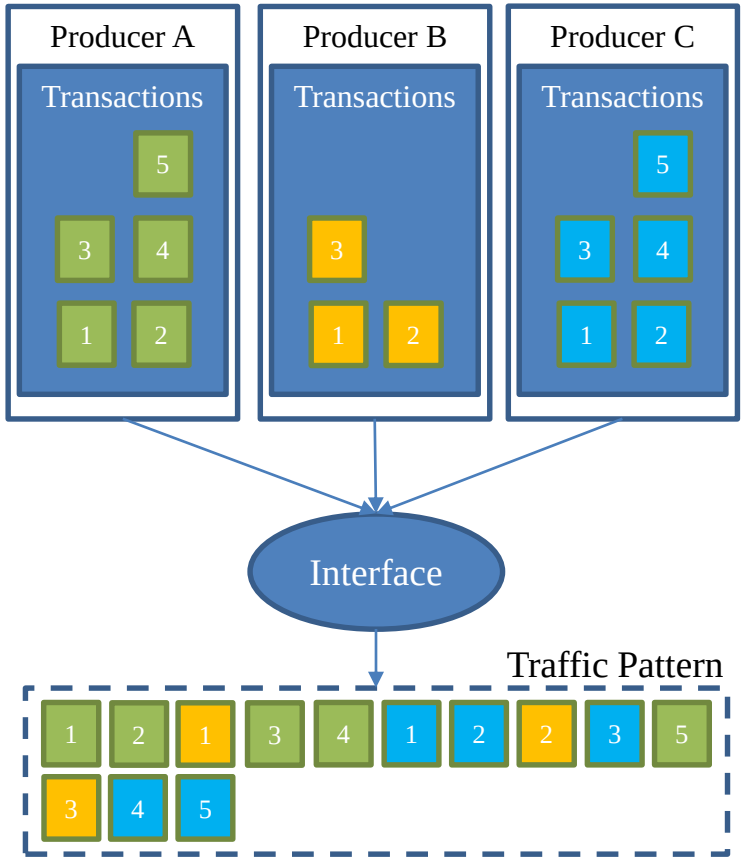
Example

- Setup
  - Hardware with an interface connected to an addressable memory
  - Interface capable of doing reads and writes
- Two buffers in the memory
  - BUF0
  - BUF1



Example

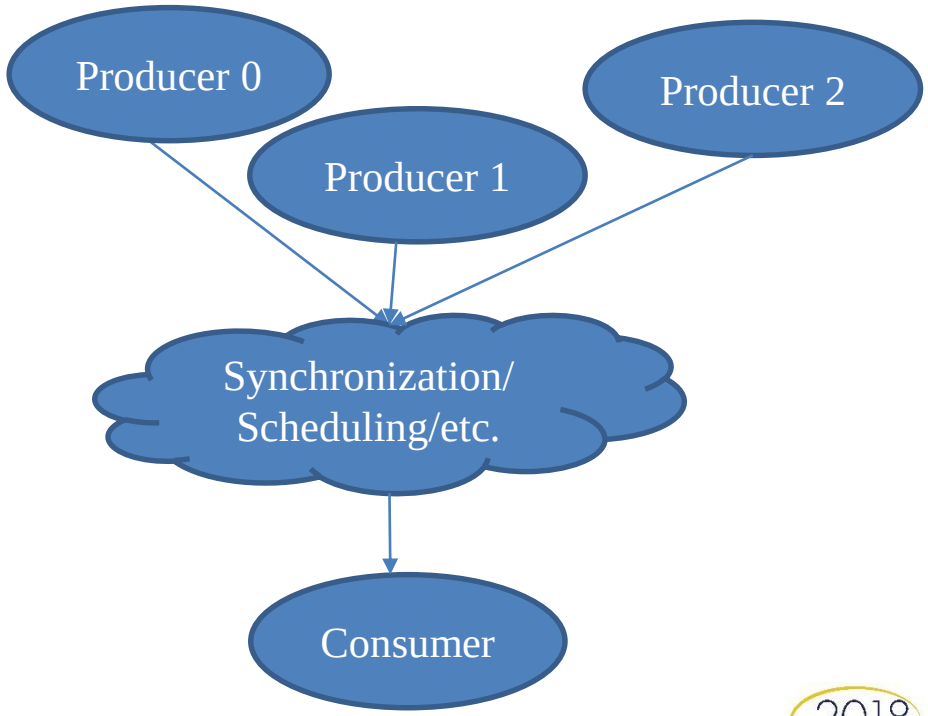
- An example of a specific traffic pattern:
  - Producer A: Reads to BUF0.
  - Producer B: Writes to BUF0.
  - Producer C: Writes BUF1
    - Starts only after producer A has done 4 transactions.



Road to an Abstraction (As always ☺)

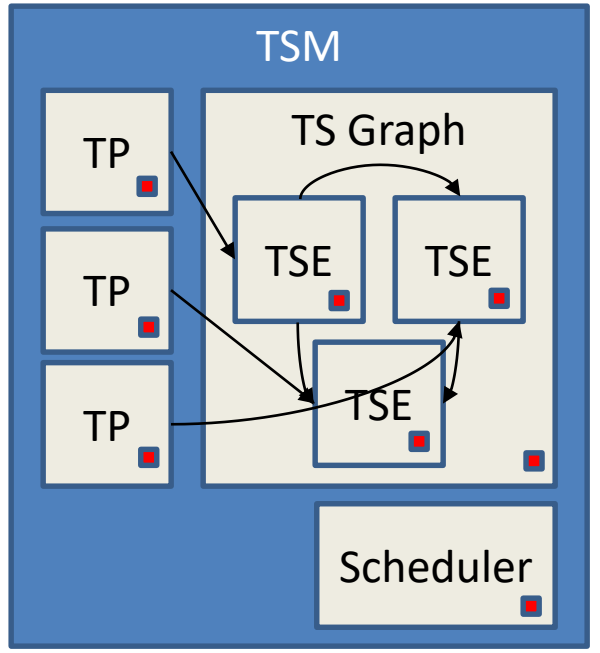
- An abstraction of the example is needed
- Provide designer with ability to express such traffic patterns

- Producer/Consumer model
  - Each producer is producing transactions
  - Consumer is the interface
  - The cloud represents
    - Synchronization (e.g. between producers)
    - Scheduling



Transaction Sequence Model

- The transaction sequence model (**TSM**) refines the producer/consumer model
- Replacing **producers** with **transaction producers** (**TP**s)
  - The **TP**s generate the transactions when requested
- Replacing “The cloud” with
  - **Transaction Sequence** graph (**TS** graph)
    - Captures synchronization etc.
    - Nodes are of the type transaction sequence elements (**TSE**s)
  - Scheduler(s)
- The bus traffic pattern at the consumer
  - Generate transactions by traversing the **TS** graph



Transaction Sequence Elements (**TSE**)

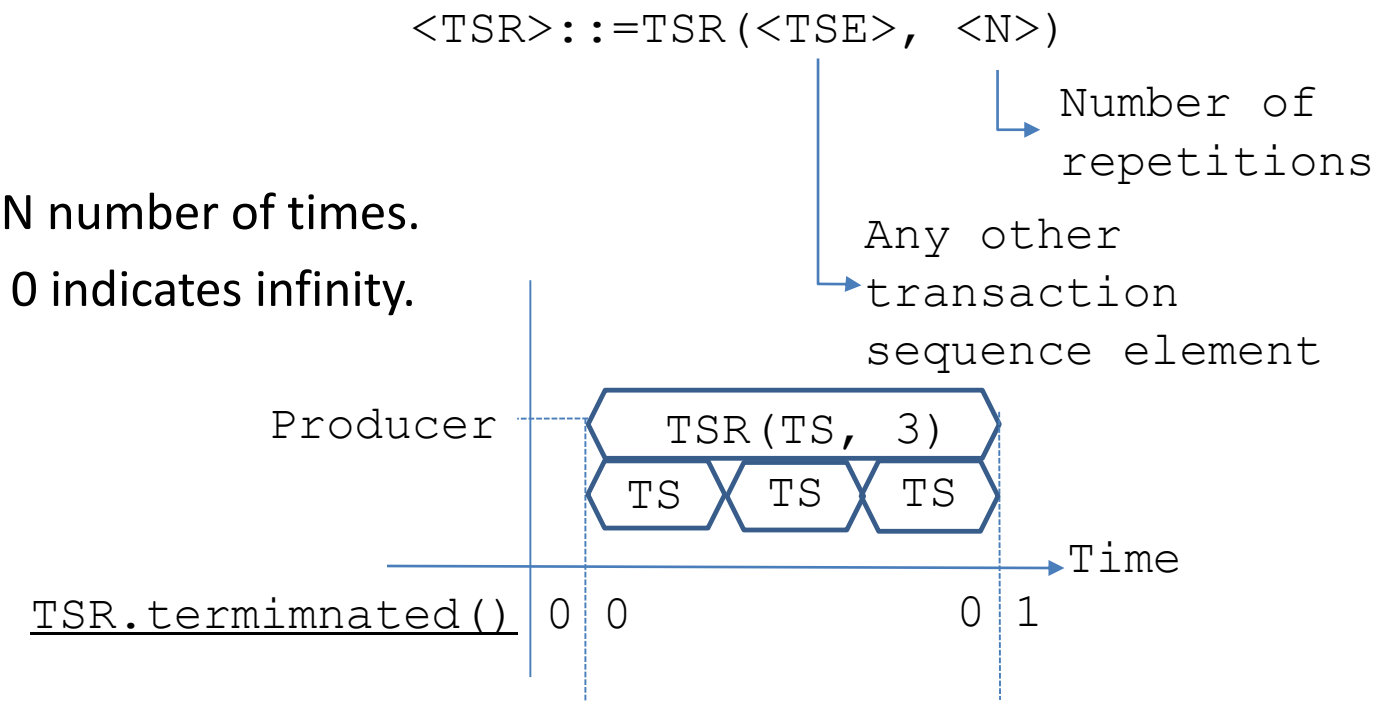
- The **TSE** nodes needs to capture the synchronization, scheduling etc.
- Node examples
  - **TST**: Request a transaction from a given producer (**TP**)
  - **TSS**: Sequencing of **TSE**s
  - **TSP**: Parallelism of **TSE**s using a scheduling algorithm
  - **TSR**: Repetition of **TSE**s
  - **TSW**: Wait for a Boolean expression to become true

- **TSE** state example
  - Number of started transactions
  - Number of ended transactions

- **TSE** function example
  - **ended**(<N>)
  - **terminated**()

TSE Example: TSR

- **tsr**(tse, N)
  - **Parameters:**
    - tse (TSE node) – Node to be evaluated N number of times.
    - N (integer) – The number of iterations. 0 indicates infinity.
  - **Return type:** Returns a TSR node.



Implementation: DSL or Not?

- One could implement a DSL capturing the TSM
  - Scanner and parser needed

- Instead
  - Specify the abstract syntax tree (AST) directly
  - Embedded DSL (eDSL)
  - Obvious benefit: no scanner and parser ☺

AST/eDSL Example

```
"""
AST/eDSL Example
"""

tp0 = tp0('tp0', 5)
tp1 = tp1('tp1', 3)
tp2 = tp2('tp2', 5)

sch    = scheduler_weight('WEIGHT', [1, 1, 1]);

tst0   = TST(tp0)
tst1   = TST(tp1)
tst2   = TST(tp2)

tsr0   = TSR(tst0)
tsr1   = TSR(tst1)
tsr2   = TSR(tst2)

tsw = TSW(tp0.ended() == 4)

tss = TSS([tsw, tsr2])

root = TSP(sch, [tsr0, tsr1, tss], tsr0.terminated() && tsr1.terminated() && tss.terminated())
```

TP definitions

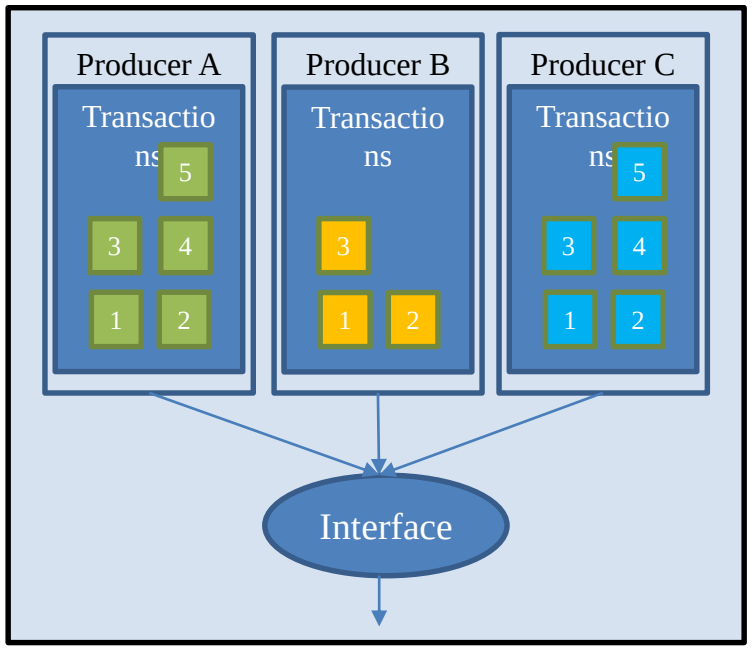
Scheduler definition
Equally weighted

TS graph reference to TPs

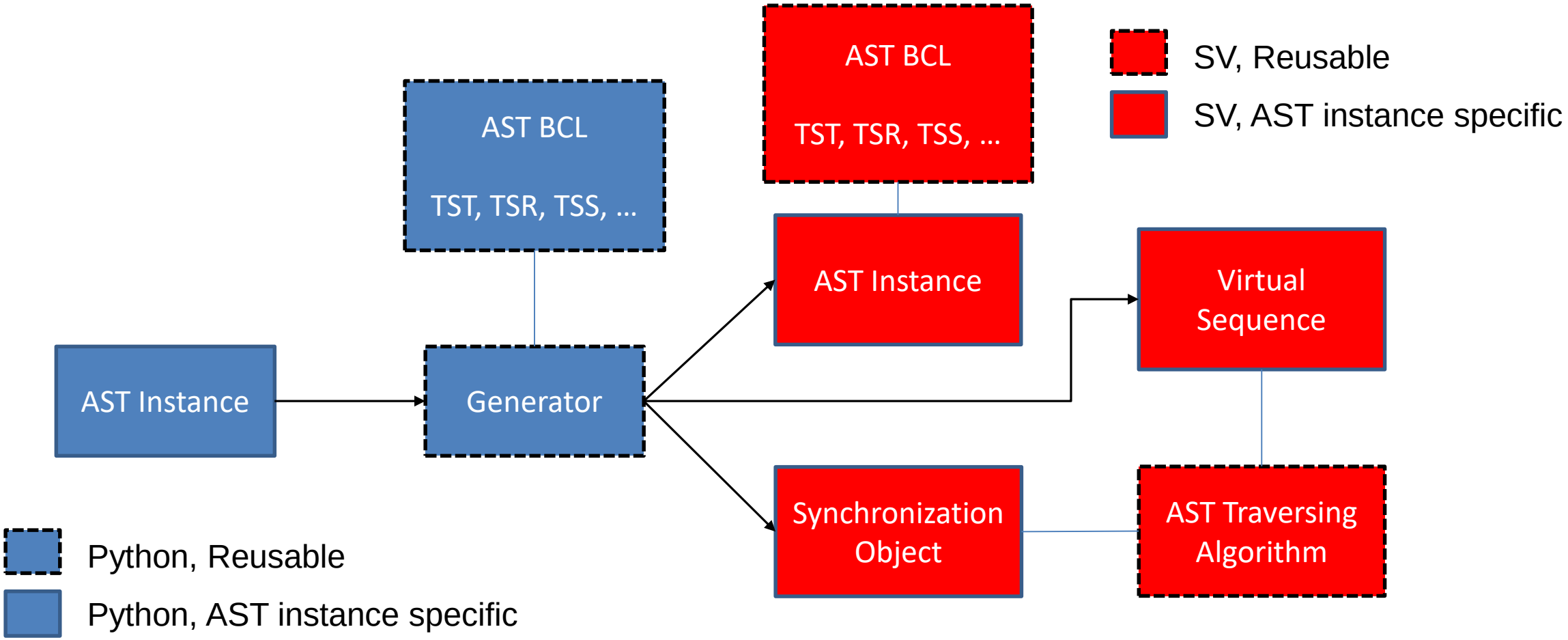
Repetition, no need for N
as the producers know their limit

Wait for tp0 to have produced 4 transactions

Root of the TS graph
with termination criteria



Implementation



Traffic Pattern Generation

- Create uvm_test which starts the generated virtual sequence
- Generated virtual sequence
  - Executes AST traversal algorithm
  - Three outcomes:
    - WAITING: The AST cannot produce a transaction. Wait until synchronizer okay's reevaluation
    - READY: Can produce the next transaction
    - TERMINATED: Done for the day
  - When READY then a transaction is provided which can be applied on the bus using the appropriate UVC
- The result should be a trace of the requested traffic pattern being applied to the bus

Conclusions

- A portable and executable specification of traffic patterns has been defined
- Using an eDSL removes a significant maintenance burden
- Using a tree traversal algorithm provides support for multiple target platforms (UVM, SystemC, etc)
- Framework can be extended to handle:
  - Multiple consumers
  - Load balancing on other aspects than number of transactions, e.g. amount of data or any other meta data (extending **TSM** and/or **TSE** state)

Questions

Finalize slide set with questions slide